Distributed storage protection in wireless sensor networks

Gianluca Dini, Lanfranco Lopriore
Dipartimento di Ingegneria dell'Informazione, Università di Pisa, via G. Caruso 16, 56126 Pisa, Italy
E-mail: {g.dini | l.lopriore}@iet.unipi.it

Abstract — With reference to a distributed architecture consisting of sensor nodes connected in a wireless network, we present a model of a protection system based on segments and applications. An application is the result of the joint activities of a set of cooperating nodes. A given node can access a segment stored in the primary memory of a different node only by presenting a gate for that segment. A gate is a form of pointer protected cryptographically, which references a segment and specifies a set of access rights for this segment. Gates can be freely transmitted between nodes, thereby granting the corresponding access permissions. Two special node functionalities are considered, segment servers and application servers. Segment servers are used for inter-application communication and information gathering. An application server is used in each application to support key management and rekeying. The rekey mechanism takes advantage of key naming to cope with losses of rekey messages. The total memory requirements for key and gate storage result to be a negligible fraction of the overall memory resources of the generic network node.

Keywords: access right, protection, sensor node, symmetric-key cryptography.

1. INTRODUCTION

We shall refer to a distributed architecture consisting of sensor nodes connected in a wireless network. In an architecture of this type, stringent limitations exist in terms of the hardware resources available in each node [15]. These limitations include the lack of hardware support for the two usual processor modes, a kernel (privileged) mode and a user (non-privileged) mode with restricted memory access, a limited memory space, and the absence of a memory management device for virtual to physical address translation [12], [20], [21]. It follows that, within the node boundaries, no separation exists between the kernel space and the user space, for instance.

In an environment of this type, we shall refer to a protection system featuring applications and segments. A *segment* is a contiguous memory area entirely contained within the boundaries of the primary memory of a single node. Segments are the basic unit of information gathering and transmission between the nodes. An *application* is the result of the joint activities of a set of cooperating nodes (the application *members*). We make no hypothesis on the activity model of each member, which can be a scheduled computation [3] or, in an event driven environment, a routine activated by a hardware interrupt [10], [24].

A classical approach to access right representation in memory is based on the concept of a *password capability* [1], [5], [16], [27]. In a segment-oriented, password-capability architecture, the protection system associates a set of passwords with each memory segment. Each password corresponds

to an access permission. A password capability is a pair (S, w) where S is a segment identifier and w is a password. If a match exists between w and one of the passwords associated with segment S, then the password capability grants its holder the corresponding access permission on S. If passwords are large and sparse, password capabilities can be freely mixed in memory with ordinary data items; an illegal attempt to modify an existing password capability (e.g. in view of an undue amplification of access permissions), or even to forge a password capability from scratch, is destined to fail, as the probability of guessing a valid password is vanishingly low.

A salient feature of password capability protection is simplicity in access right distribution. A process that holds a valid password capability can grant the corresponding access rights to another process by a simple action of password capability copy, from its own address space to the address space of the recipient process. In turn, the recipient process may well transmit the password capability to a third process. In a situation of this type, it is hard, if not impossible, to keep track of all copies of a given password capability that exist in the system at the same given time. This exacerbates the problem of access right revocation: the original owner of a given password capability should be in a position to retract the password capability from each subsequent recipient, selectively. Of course, if we modify the passwords of a given segment, we revoke all the password capabilities referencing that segment. This revocation mechanism cannot be used for selective revocation of a subset of all the password capabilities for the same given segment.

In this paper, we shall refer to a variant of the password capability model that has been designed to comply with the resource limitations, outlined above, which characterize the sensor nodes in a wireless sensor networks. In our approach, within a node, every software routine has unlimited access to the whole primary memory of that node, irrespective of segment boundaries; whereas a routine running in a given node can access a remote segment stored in the primary memory of a different node only by presenting a *gate* for that segment. A gate is a form of password capability protected cryptographically, which references a segment and specifies a set of access rights for this segment. Possible access rights are read, write, or both read and write. Gates are protected from tampering by a form of symmetric-key cryptography [13], [33], superior to public key cryptography in both terms of low computation requirements and low energy costs [4], [17].

In a sensor node, the high memory cost of a set of passwords for each memory segment is not acceptable. Our gate implementation uses a single set of password for each node. We have obtained this result by taking advantage of cryptography to incorporate the name of the segment referenced by a given gate into the protection field of this gate. A small set of system primitives, the *protection primitives*, makes it possible to define segments, to generate gates for existing segments, and to use gates in remote segment accesses. A node that generates a gate is free to transmit this gate to another

node, thereby granting the corresponding access permissions to the recipient node. Two or more segments can be defined for the same memory area. By deleting one of these segments, we revoke the gates referencing this segment; revocation does not affect validity of the gates for the remaining segments.

The rest of this paper is organized as follows. Section 2 introduces our protection environment with special reference to segments and gates. The protection primitives are presented, and the actions involved in the execution of each primitive are illustrated with special reference to interactions between nodes. Section 3 presents our application model. Two special node functionalities are introduced, the *application server*, used within the application boundaries to support information gathering, key management and rekeying, and the *segment server*, used for inter-application communication. Section 4 discusses the motivations for the proposed organization from a number of salient viewpoints, including the hardware limitations existing in sensor nodes, gate manipulation and revocation, security, and the memory requirements for key and gate storage. We consider two different network topologies in special depth, a configuration featuring a form of full pairwise connectivity at the application level, and a hierarchical topology featuring a *general server* that gathers data from all the application servers. Relations of our work to previous works are outlined. Section 5 gives concluding remarks.

2. THE PROTECTION MODEL

2.1. Segments

In the previous section, we have defined a segment as a contiguous memory area that is entirely contained within the boundaries of the primary memory a single node. A segment S is identified by pair $S = (M, C)$ where M is the node storing S, and C is the *local identifier* of S in M. In node M, a table, the *segment table* ST_M, contains the associations of local segment identifiers with the corresponding areas in the primary memory of that node. The table entry for local segment C, i.e. segment $S = (M, C)$, contains the starting address B of this segment in the primary memory of M (the segment *base*) and the segment *length* L.

2.2. Gates

A set P_M of three *passwords*, $P_M = \{p_\mathrm{R}, p_\mathrm{W}, p_\mathrm{RW}\}$, is associated with each given node M and is stored in the primary memory of this node. Each password corresponds to an access right for the segments in M. Password p_R corresponds to access right R, which makes it possible to access the segments for read. This is similar to password p_W for access right W, which makes it possible to access the segments for write, and to password p_RW for access right RW, which makes it possible to

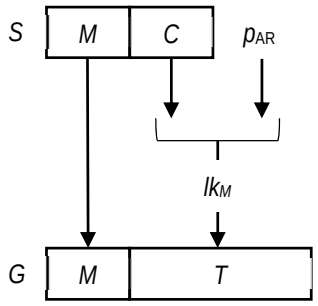

Figure 1. Generation of gate $G = (M, T)$ that specifies access right AR for segment $S = (M, C)$. p_{AR} is the password that corresponds to AR in the password set P_M associated with node M.

access the segments for both read and write.

A gate G referencing segment $S = (M, C)$ in node M is a pair $G = (M, T)$, where T is a *protection field* that includes the specification of the segment local identifier C and a password p. If a match exists between p and one of the passwords in P_M, then the gate grants the corresponding access right for segment S. Quantity M is in plaintext, whereas quantity T is encrypted by using a symmetric-key cipher and a cryptographic key, called the *local key* lk_M, which is associated with node M. lk_M is stored in node M; it is never transmitted or revealed by M to any other node, and it is exclusively aimed at encrypting the gates for the segments in M.

Figure 1 shows the generation of gate $G = (M, T)$ granting access right AR for segment $S = (M, C)$, AR being one of R, W or RW. Let p_{AR} denote the password in P_M that corresponds to this access right. Quantity T is the result of encrypting pair (C, p_{AR}) by using a symmetric key cipher and local key lk_M. Thus, a gate referencing a segment in node M can only be assembled in this node, as gate generation requires knowledge of local key lk_M. Throughout this paper, we assume that ciphers comply with an encryption mode supporting both authentication and confidentiality, e.g. the Counter with CBC-MAC (CCM) mode [11].[1]

Figure 2 shows the reverse transformation of gate G into plaintext. Local key lk_M is used to decrypt the protection field T and obtain quantities C and p. Quantity p is compared with the passwords in P_M to validate the result of the transformation. If a match is found and p_{AR} is the matching password, validation is successful, gate G references segment $S = (M, C)$ and specifies the access right corresponding to p_{AR}. Thus, a gate referencing a segment in node M can only be disassembled in M, as the decryption process uses the local key lk_M of this node.

[1] Intuitively, a single encryption key can be used for both authentication and confidentiality. The sender authenticates the header and the payload, it appends the resulting Message Identification Code (MIC) to the payload and, finally, it encrypts the bundle. The receiver decrypts the ciphertext into a payload and a MIC, and verifies the MIC against the received header and payload.

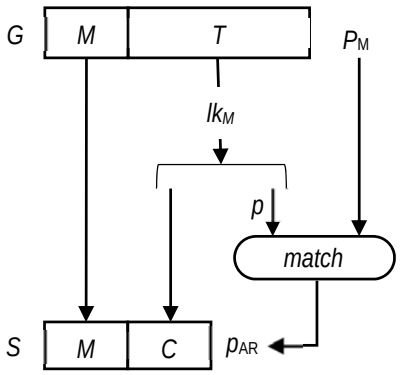

Figure 2. Validation of gate $G = (M, T)$ and subsequent transformation into segment identifier $S = (M, C)$. Gate G specifies the access right corresponding to password p_{AR}.

2.3. Protection primitives

Table I shows the set of the protection primitives that form the interface of the protection system to applications. Execution of a protection primitive is entirely confined within the boundaries of the node where this primitive has been issued (the *current* node), or, for a few primitives, it produces node interactions across the network. Interactions take the form of message exchanges. A message consists of a *header* and a *body*. The header is in plaintext, and the body is encrypted using a symmetric-key cipher. Besides the control information, necessary, for instance, for message routing, the header contains the *name* of the key that was used to encrypt the body. A message can be successfully sent from node *M* to node *N* encrypted by a given key only if both nodes holds the *value* of this key. Node *N* will use the key name, as specified in the message header, to identify the correct key and decrypt the message body.

In the following, the actions caused by execution of each protection primitive will be described in detail. We shall use the term *local segment* to denote a segment allocated in the primary memory of the current node, whereas a *remote segment* is a segment allocated in the primary memory of a different node (a *remote node*).

2.3.1. Allocating and deleting segments

The processor of a given node can freely access the whole primary memory of that node for both read and write, irrespective of segment boundaries. No form of protection is enforced on these local memory accesses; in particular, a gate is not required to access a local segment. On the other hand, an access to a remote segment can be successfully accomplished only if a gate for that segment is presented to the remote node where the segment is allocated. Two actions are defined on a remote segment: to read the segment contents and to replace these contents. These actions are made possible by a gate specifying permission to read (i.e. either the R or the RW access right) or permission to

Table I. Protection primitives.[1]

C ← *newSegment*(*B*, *L*) Allocates a new segment of length *L* that starts at address *B* of the primary memory of the current node. Returns the local identifier *C* of this segment.
G ← *newGate*(*C*, *AR*) Returns a gate *G* specifying access right AR for the segment whose local identifier is *C*. AR is one of R, W or RW. Gate *G* is generated by using the local key of the current node.
deleteSegment(*C*) In the current node, deletes the segment whose local identifier is *C*.
readSegment(*k*, *G*, *addr*) Copies the contents of the remote segment referenced by gate *G* into a memory area starting at address *addr* of the primary memory of the current node. Uses key *k* to communicate with the node where the remote segment is stored. *G* should specify permission to read (access right R or RW).
writeSegment(*k*, *G*, *addr*) Replaces the contents of the remote segment referenced by gate *G* with quantities taken from a memory area starting at address *addr* of the primary memory of the current node. Uses key *k* to communicate with the node where the remote segment is stored. *G* should specify permission to write (access right W or RW).

[1] The *current node* is the node where the given protection primitive is issued.

write (i.e. either the W or the RW access right), respectively.

Suppose that node *N* should be allowed to access an area in the primary memory of a different node *M*. To this aim, node *M* allocates a local segment corresponding to that area, it generates a gate for this segment and transmits a copy of this gate to node *N*. Segment allocation and gate generation are made possible by protection primitives *newSegment*() and *newGate*().

In detail, execution of the *C* ← *newSegment*(*B*, *L*) primitive in node *M* allocates a new segment *S* = (*M*, *C*) in the primary memory of this node and returns the local identifier *C* of this segment. Argument *B* specifies the base of the new segment, and argument *L* specifies the segment length. Execution of this primitive generates local segment identifier *C*, first. Then, an entry of segment table ST_M of node *M* is reserved for the new segment; the base and length fields of this entry are filled with quantities *B* and *L*. A simple strategy for the generation of local segment identifiers is a sequential generation, which can be implemented as follows: in each node, a *segment counter* contains the local identifier of the segment to be allocated next in that node; after segment allocation, the contents of the segment counter are incremented by 1. It should be noted that *newSegment*() does not prevent segments to overlap. This means that the same storage area may be part of two or more segments.

If executed in node *M*, the *G* ← *newGate*(*C*, AR) protection primitive returns a new gate *G* granting access right AR for segment *S* = (*M*, *C*), AR being one of R, W or RW. Let P_M be the set of passwords associated with node *M*, and let p_{AR} be the password in P_M that corresponds to access right AR. Execution of this primitive uses local key lk_M of node *M* to encrypt pair (*C*, p_{AR}) and form quantity *T* (see Figure 1). Then, node name *M* is paired with quantity *T* to form gate *G* = (*M*, *T*). It should be noted that the aim of *newGate*() is restricted to the generation of gates for segments allocated in

the primary memory of the current node, as it requires the local key. Thus, a node cannot generate a gate for a segment in a different node.

Finally, execution in node M of the *deleteSegment*(C) protection primitive deletes segment $S = (M, C)$. Execution accesses the segment table ST_M of this node and deletes the entry reserved for this segment. *deleteSegment*() does not modify the contents of the primary memory area reserved for the segment to be deleted. This means that if we define two or more segments for the same memory area, and then we delete one of these segments, deletion has no impact on the other segments.

2.3.2. Accessing remote segments

Let $G = (N, T)$ be a gate that references segment $S = (N, C)$ stored in node N, and let L be the length of this segment. Furthermore, let p be the password specified by the protection field T of G, and let P_N be the set of the passwords associated with N. If executed in node M, protection primitive *readSegment*(k, G, *addr*) copies the contents of segment S from N into a memory area that starts at address *addr* of the primary memory of M. Argument k is a cryptographic key that is used for communication between M and N. Execution terminates successfully only if password p matches one of the passwords in P_N, and the matching password grants permission to read (i.e. it is either the p_R or the p_{RW} password). Execution of this primitive is as follows:

1. Node M sends a message to node N asking for a "number used once" (*nonce*) [33].
2. Node N generates a random number E_N to be used as a nonce, and sends it back to node M.
3. Node M generates a nonce E_M and assembles a message m containing gate G, nonce E_N and nonce E_M. The message is sent to node N in ciphertext, and the encryption key is k.
4. Node N uses key k to decrypt message m into triple (G, E',E''), and validates nonce E' by verifying that $E' = E_N$. Then, node N uses its own local key lk_N to decrypt the protection field T of gate G into pair (C, p) (see Figure 2). Quantity p is compared with passwords p_R and p_{RW}; if a match is found, gate G is valid (it specifies permission to read).
5. If nonce validation fails, or gate validation fails, node N returns a message including nonce E'' and a negative reply to node M, and execution of *readSegment*() terminates with failure; otherwise
6. Node N assembles a message containing nonce E'', the contents of segment $S = (N, C)$ and a positive reply. This message is sent to node M in ciphertext, and the encryption key is k.
7. Node M uses key k to decrypt the message into pair (E'', S), and validates nonce E'' by verifying that $E'' = E_M$. Then, M stores the contents of segment S into a memory area of length L starting at address *addr* of its own primary memory.

Nonces E_N and E_M are aimed at preventing forms of replay attacks [33]. They allow the corresponding node to distinguish a new request/reply from an illicit replay of a previous request/reply. In step 3, in the message to node *N*, the gate should be indissolubly linked to the nonces, and similarly, in steps 5 and 6, in the message to node *M*, the operation results should be indissolubly linked to the nonce. In steps 3 and 6, the name of the key used to encrypt the message is specified in the message header; if the recipient node does not hold the value of this key, execution terminates with failure.

Let $G = (N, T)$ be a gate that references segment $S = (N, C)$ stored in node *N*, let *L* be the length of this segment, and let *p* be the password specified by the protection field *T* of *G*. Furthermore, let P_N be the set of passwords associated with *N*. If executed in node *M*, protection primitive *writeSegment*(*k, G, addr*) copies the contents of a memory area of length *L* that starts at address *addr* of the primary memory of *M* into segment *S*. Argument *k* is a cryptographic key that is used for communication between *M* and *N*. Execution terminates successfully only if password *p* matches one of the passwords in P_N and the matching password grants permission to write (i.e. it is either the p_{W} or the p_{RW} password). The actions caused by execution of *writeSegment*() are as follows:

1. Node *M* sends a message to node *N* asking for a nonce.
2. Node *N* generates a nonce E_N and sends it back to node *M*.
3. Node *M* generates a nonce E_M and assembles a message *m* including gate *G*, nonce E_N, nonce E_M and the contents *a* of a memory area of length *L* that starts at address *addr* of the primary memory of *M*. The message is sent to node *N* in ciphertext, and the encryption key is *k*.
4. Node *N* uses key *k* to decrypt message *m* into quadruple (*G*, *E'*, *E"*, *a*), and validates nonce *E'* by verifying that $E' = E_N$. Then, *N* uses its own local key lk_N to decrypt the protection field *T* of gate *G* into pair (*C*, *p*). Quantity *p* is compared with passwords p_{W} and p_{RW}; if a match is found, gate *G* is valid (it specifies permission to write).
5. If nonce validation fails, or gate validation fails, node *N* returns a message including nonce *E"* and a negative reply to node *M*, and execution of *writeSegment*() terminates with failure; otherwise
6. Node *N* replaces the contents of segment $S = (N, C)$ with quantity *a*. Then, *N* assembles a message containing nonce *E"* and a positive reply; this message is sent to node *M* in ciphertext, and the encryption key is *k*.
7. Node *M* uses key *k* to decrypt the message into pair (*E"*, *S*), and validates nonce *E"* by verifying that $E" = E_M$.

3. APPLICATIONS

As anticipated in Section 1, an application is the result of the joint activities of a set of nodes, the

application members, which cooperate in the same task. All the members share a cryptographic key, which is called the *application key*. This key is used in the communications between the members, which are carried out by taking advantage of protection primitives *readSegment*() and *writeSegment*().

Let us refer, for instance, to nodes M and N, and suppose that both of them are members of application A. Let k_A be the key of this application. In a symmetric communication paradigm, node M reserves a segment S_1 for the information transfers from M to N. A gate $G_1 = (M, T_1)$ specifying access right R for S_1 is granted to N; the protection field T_1 of this gate is encrypted by using the local key lk_M of node M. Symmetrically, node N reserves a segment S_2 for information transfers from N to M. A gate $G_2 = (N, T_2)$ specifying access right R for S_2 is granted to M; the protection field T_2 of this gate is encrypted by using the local key lk_N of node N. Let $addr_M$ be the address of a memory area in node M, and similarly for $addr_N$ in node N. M transmits an information item to N by writing this information item into S_1; this action is not mediated by the protection system, as S_1 is part of the primary memory of M. In turn, N copies the contents of S_1 into its own primary memory by executing primitive *readSegment*(k_A, G_1, $addr_N$). Symmetrically, N transmits an information item to M by writing this information item into S_2; in turn, M reads the contents of S_2 by issuing primitive *readSegment*(k_A, G_2, $addr_M$).

In a different, asymmetrical communication approach, node M allocates a segment S in its own primary memory and reserves this segment for communication with node N. Node N is granted a gate $G = (M, T)$ specifying access right RW for S; the protection field T of this gate is encrypted by using local key lk_M of node M. Node M accesses S directly, whereas node N reads and modifies the contents of S by executing *readSegment*(k_A, G, $addr_N$) and *writeSegment*(k_A, G, $addr_N$).

3.1. Inter-application communications

A sensor network may host several applications. Point-to-point communication and information sharing between two members of different applications take advantage of a cryptographic key, called a *nonlocal key*, shared by these applications. Let M_1 be a node of application A_1, and let M_2 be a node of application A_2. M_1 and M_2 share a nonlocal key that we shall denote by nk. Let $addr_1$ be the address of a memory area in M_1, and similarly for $addr_2$ in M_2. In a symmetric communication paradigm, M_1 reserves a segment S_1 for the information transfers from M_1 to M_2. Node M_2 holds a gate G_1 specifying access right R for S_1. M_1 accesses S_1 for write directly, and M_2 accesses this segment for read by issuing primitive *readSegment*(nk, G_1, $addr_2$). Symmetrically, M_2 reserves a segment S_2 for the information transfers from M_2 to M_1. M_2 accesses S_2 for write directly; M_1 holds a gate G_2 specifying access right R for S_2 and issues *readSegment*(nk, G_2, $addr_1$) to access S_2 for read. Thus, the messages transmitted between M_1 and M_2 are encrypted by using nonlocal key nk, which is only held by these

two nodes. It follows that any other node, e.g. an intermediate node in the routing path between M_1 and M_2, will not be able to decrypt these messages, as it does not possess the key.

Of course, the inter-application communication paradigm, illustrated above, can be extended to an arbitrary number of applications, by reserving a nonlocal key for communication between each application pair. If the nodes of each application are structured hierarchically, only the node at the highest level in the hierarchy will be deputed to inter-application communication, for instance.

3.1.1 Segment servers

A different, centralized paradigm of communication and information sharing between different applications takes advantage of a node, called the *segment server*, which is not part of these applications. Let V be a node acting as a segment server for applications A_0, A_1,..., and let M_i be a node of application A_i. Node M_i shares a nonlocal key nk_i with server V. Let $addr_i$ be the address of a memory area in M_i. Server V reserves a segment S_i for the information transfers with M_i, and M_i holds a gate G_i specifying access right RW for S_i. M_i accesses S_i for read by issuing primitive *readSegment*(nk_i, G_i, $addr_i$), and it accesses S_i for write by issuing primitive *writeSegment*(nk_i, G_i, $addr_i$).

In this inter-application communication paradigm, only those nodes that are actually involved in information sharing need to have access to a server. If the nodes of each given application are structured hierarchically, only the node at the highest level in the hierarchy needs to interact with the server, for instance.

3.2. Key management

Application keys are subject to be replaced. This is a peculiar aspect of wireless sensor networks [34], [38]. In the *periodic rekeying* approach [19], [29], [31], keys are renewed at regular intervals to maintain resilience to attacks and failures, and safeguard secrecy. In a given application A, a rekey will be necessary when a node leaves A, to prevent that node from taking advantage of the old key any longer (*forward secrecy* [6], [7], [9]). Furthermore, consider a node N in the routing path between two nodes of A and suppose that this node is not part of A. Of course, N can gather the messages exchanged between these two nodes, but it cannot access the contents of these messages (they are encrypted by using application key k_A, and N does not possess this key). Now suppose that N is added to application A: a rekey will be necessary to prevent N from deciphering the old messages (*backward secrecy* [6], [7], [9]).

Our approach to application key management is as follows. In each application, a node, called the *application server*, is responsible for the distribution of a new application key to all the nodes that are members of that application, when a rekey takes place. As anticipated in Section 2.3, each application key has a numeric *name* and a *value*. The application server generates the names of the keys

of its own application in sequence, so that, numerically, the name of a given key is always greater than the name of a previous key. A simple strategy for key name generation takes advantage of a counter, which we call the *key counter*, maintained in the primary memory of the application server. The key counter is initialized to 0 and is incremented by 1 after generation of a new application key. The name of the new application key is a bit string featuring the binary representation of the name of the application server in the most significant positions, and the contents of the key counter in the least significant positions.[2]

Key replacement takes advantage of a segment that the application server reserves in its own primary memory for each node that is a member of that application. This segment is called the *key repository*. Each member holds a gate granting access right R for its own key repository. When the application key should be replaced, the application server inserts the name and the value of the new application key into the repository of each member, and then sends a *rekey message* to all the members. Consequently, each member uses the gate for its own repository to read the new application key. Message transmission between the member and the application server takes advantage of a nonlocal key.

In more detail, suppose that node M is a member of application A, let AV be the application server of this application, let RP_M be the key repository that AV has reserved for M in its own primary memory, and let nk_M be the nonlocal key shared by M and AV. Node M holds a gate $G = (AV, T)$ specifying access right R for segment RP_M (the protection field T of gate G is encrypted by using local key lk_{AV} of AV). Now suppose that the current key of application A should be replaced by a new key. Application server AV inserts both the name and the value of the new key into the repository of each member, and then it sends a rekey message to all the members. On receipt of the rekey message, node M issues protection primitive *readSegment*(nk_M, G, *addr*), where *addr* is the address of a primary memory area in M. Execution of this primitive uses nonlocal key nk_M to communicate with AV; the name and the value of the new key are copied from RP_M into the memory area at address *addr*. Node M will use these information items to update its own copy of the application key, so that from now on M will use the new key.

As seen previously, when a node M leaves its own application A, it is necessary to change the application key; the new key will be distributed to all the nodes in A except M. A result of this type

[2] This key naming approach can be easily extended to local and nonlocal keys, as follows. The name of the local key of a given node is a bit string featuring the binary representation of the name of this node in the most significant positions, and all 1s in the least significant positions. Decreasing key names starting from the local key name will be reserved to the nonlocal keys.

will be obtained by taking advantage of our rekeying mechanism, as follows. The server *AV* of application *A* writes the new key into the key repository of all the members of *A* except *M*. Then, *AV* sends a rekey message; consequently, each member executes primitive *readSegment*() to read the new key from its own key repository, and then it update the key. It should be noted that, if the key replacement message reaches node *M* and this node executes *readSegment*(), this action produces no other effect, as the key repository RP_M of this node still contains the old, discarded key.

3.2.1. Rekey messages

Let us refer to nodes *M* and *N*, and suppose that they are both members of application *A*. Suppose that *M* sends a message to *N*. As seen in Section 2.3, a message consists of a header in plaintext and a body in ciphertext; the header contains the name of the key that was used to encrypt the body. On receipt of the message from *M*, node *N* compares the name of the application key k'_A used to encrypt the message, as specified by the message header, with the name of its own application key k_A. If $k_A = k'_A$, then message transmission and delivery are successful (*N* can decrypt the message). If this is not the case, we will take advantage of the fact that, numerically, the name of a key of a given application is always greater than the name of a previous key of that application. Thus, if $k_A > k'_A$, key k'_A is outdated. This means that either *M* encrypted the message before updating the key, or a rekey message was lost, due to a network fault [19], for instance, or *M* is no longer part of application *A*, so it does not participate in the rekey. Node *N* cannot discriminate between situations of this type, so it discards the message and sends a negative reply to *M*. Consequently, *M* updates the key by reading the new key from its own key repository, and then sends the message again (of course, if *M* is no longer part of application *A*, the repository still contains the old key, and any attempt to update the key is destined to fail). Finally, if $k_A < k'_A$, node *N* holds a key older than that used by *M* to encrypt the message. *N* will update the key by reading the new key from its own repository. Afterwards, *N* will be in a position to decrypt the message.

We may conclude that our rekey mechanism is able to cope with losses of rekey messages, which produce no negative consequence for the communication ability of the nodes involved. This feature is especially important for reliable application rekeying in an unreliable network environment [19], [22].

4. DISCUSSION

4.1. Hardware limitations

As seen in Section 1, in a sensor node the absence of a memory management device for virtual to physical address translation, and the lack of hardware support for the two processor modes, the

kernel (privileged) mode and the user (non-privileged) mode, implies that no address space separation exists between a system routine, e.g. a protection primitive, and an application routine. Any piece of software running in a given node has unlimited access to the whole primary memory of that node. It is not viable to confine cryptographic keys and passwords to protected memory areas, for instance; instead, every software routine can read (and possibly modify) all these critical information items. It follows that each node can host a single application. Indeed, in a node that contains two applications, a routine is in a position to use the keys of both these applications for message decryption; this is a security hole we are aimed at avoiding (the routine may reveal the messages exchanged by the members of one application to the members of the other application, thereby breaking the application boundaries).

Furthermore, it is virtually impossible to enforce gate protection on local segment accesses; instead, gates are only effective across node boundaries. Consider, for instance, a segment *S* defined in the primary memory of node *M*. Any software routine in *M* will be able to access the primary memory area corresponding to this segment, and this access is not subject to any form of access right control. On the other hand, a different node *N* will have to ask *M* for cooperation to access *S*, by executing protection primitives *readSegment*() and *writeSegment*(), and presenting a gate for *S*. Indeed, access control and gate-based protection is enforced by the physical separation of the address spaces of the two nodes.

4.2. Cryptographic keys

4.2.1. Local keys

Our protection system takes advantage of three types of cryptographic keys: local keys, nonlocal keys and application keys (Table II). A local key is used in each node to encrypt the gates for the segments in the primary memory of that node, as occurs in the execution of the *newGate*() protection primitive (see Section 2.3.1). The local key is never transmitted across the network, and it is never revealed to a node different from the original owner. It follows that, if a node generates a gate *G* and grants this gate to a different node, the recipient node will not be able to decipher *G* and alter its contents, e.g. to replace the password and amplify the access rights.

4.2.2. Nonlocal keys

Nonlocal keys are used for communication between a node and its servers. Our system defines two types of servers, segment servers and application servers. As seen in Section 3.1.1, communication between a segment server and each of its clients takes advantage of a nonlocal key. This key allows secure message exchange, so that a node in the routing path between the server and that client

Table II. Cryptographic keys.

Local key A local key is used in each node to encrypt the gates for the segments in that node.
Nonlocal key A nonlocal key is used in each node to communicate with the application server. If the node is connected with a segment server, a further nonlocal key is reserved for communication with this server. Nonlocal keys are also used for point-to-point communication between the members of different applications.
Application key An application key is reserved in each application for communication between the members of that application.

cannot decipher the messages. A nonlocal key is used instead of an application key, as we cannot allow a single node, the segment server, to possess more than a single application key.

As seen in Section 3.2, application servers are deputed to key distribution as part of the key replacement activities. In the given application, the application server reserves a key segment for each member of that application. Communication between the application server and the member takes advantage of a nonlocal key. The member uses this nonlocal key to execute *readSegment*() and read the new application key from the key segment when a rekey takes place. The application server is itself part of the application; utilization of a nonlocal key instead of the application key is motivated as follows. Consider a node that has been evicted from the application; a rekey should take place to prevent that node from taking advantage of the old key any longer. If the new application key is transmitted in a message encrypted by using the old key, and the evicted node is in a position to capture the message, it will be able to decrypt the message and obtain the new key.

Finally, as seen in Section 3.1, a nonlocal key is also used for point-to-point communication between two nodes of different applications. In this case, too, it is not viable to take advantage of the application key of one of the two nodes, as this would imply that the other node holds two application keys.

4.2.3. Application keys

An application key is reserved in each application for communication between the application members. Let *M* and *N* be two members of application *A*. When a message is exchanged between *M* and *N*, any member of application *A* in the routing path between *M* and *N* is in a position to decrypt the message; this is not a security hole, as the members of the same application are considered mutually trustworthy. On the other hand, a member of a different application will not be able to read the message contents, as it does not possess the encryption key.

4.3. Memory requirements

Owing to the stringent limitations concerning the memory resources available in a sensor node

[30], [35], the memory requirements for key and gate storage are significant factors. As seen in Section 3.2, a key has a name and a value. For an application key, the key name consists of a node name (that of the application server) followed by the value of a key counter; local and nonlocal keys present a similar structure and identical memory requirements. In a large network featuring up to 2^{16} nodes, the size of a node name is 16 bits, and a 32-bit key name will allow a substantial number of rekeying actions. If the size of a key value is 128 bits, we have a key size of 20 bytes.

As seen in Section 2.2, a gate consists of a node name, in plaintext, and a protection field, in ciphertext. The protection field includes a local segment identifier and a password. If the size of a node name is 16 bits, and the primary memory of a node is up to 64 Kbytes, for 128-bit passwords we have a protection field of 18 bytes, and the size of a gate is 20 bytes.

We wish to point out that, in a given gate, the node name is not involved in the transformation of the gate from plaintext into ciphertext. Indeed, knowledge of the node name is only necessary in the execution of the *readSegment*() and *writeSegment*() protection primitives to identify the network position of the referenced segment. It follows that the location of the node name field in memory is irrelevant. A node holding a collection of gates referencing segments in the same remote node may well maintain a single copy of the node name, for instance; the node will reconstruct the association of the name of the remote node with the protection field when needed, just before issuing *readSegment*() or *writeSegment*().

4.3.1. Hierachical configuration

As seen in Section 3.2, the members of the generic application are structured hierarchically. A member assumes the role of the application server; it reserves a key repository for each of the other application members. Key repositories are used when a rekey takes place, for distribution of the new key. We shall now extend this model to cope with application data gathering. Besides a key repository, the application server reserves a *data repository* for each member. Each member holds a gate allowing write access to its own data repository. The member uses the *writeSegment*() protection primitive to deposit the results of its own computations into the data repository. Execution of *writeSegment*() takes advantage of the same nonlocal key that is used for rekeying.

The hierarchical member structure existing within the application boundaries can be extended at the network level by introducing a *general server* to which all application servers are connected. The general server reserves a segment in its own primary memory for each application server. This segment is called the *application repository*. Each application server holds a gate allowing write access to its own application repository. The application server uses the *writeSegment*() primitive to deposit the results of the computations of the corresponding application, as follow from the activity of the

members, into its own application repository. A nonlocal key is reserved for point-to-point communication between the application server and the general server.

In this hierarchical model, with reference to the generic application, each member holds three keys: (i) a local key, used for gate encryption; (ii) a nonlocal key, used to access both the key repository and the data repository in the application server; and (iii) an application key, used to communicate with the other application members. Let n denote the number of nodes that form the sensor network, let a denote the number of applications in which these nodes are partitioned, and let $v = n / a$ denote the average number of nodes in each application. An application server holds the following keys: (i) a local key, used for gate encryption; (ii) an average of v - 1 nonlocal keys, one key for each of the other application members, used to communicate with that member; and (iii) a nonlocal key, used to communicate with the general server. Thus, the application server holds a total of $v + 1$ keys. For $n = 1024$ nodes and $a = 16$ applications, we have $v = 64$, for instance. In this network configuration, an application server holds 65 keys on average, with a total memory requirement of 1300 bytes.

Furthermore, each application member holds one gate for the key repository, used to support the rekey process, and one gate for the data repository, used to deposit the computation results. The application server holds a single gate referencing the application repository in the general server.

4.3.2. Full pairwise connectivity at the application level

In a different approach, we support a form of full pairwise connectivity [32] at the level of the application servers, as follows: each application server reserves an application repository for each of the other application servers; a nonlocal key is shared by the two application servers for data exchange, which occurs via *writeSegment*().

In this network configuration, the generic member of each application holds three keys: (i) a local key, used for gate encryption; (ii) a nonlocal key, used to communicate with the application server; and (iii) an application key, used to communicate with the other application members. The application server holds the following keys: (i) a local key, used for gate encryption; (ii) an average of v - 1 nonlocal keys, one key for each of the other application members, used to communicate with that application member; and (iii) a - 1 nonlocal keys, one key for each of the other application servers, used to communicate with that application server. Thus, in each application server, we have a total of $v + a$ - 1 keys. For instance, if $n = 1024$ nodes and $a = 16$ applications, the average number of nodes for each application is $v = 64$, and an application server holds 79 keys on average, with a total memory requirement of 1580 bytes.

Furthermore, each application member holds one gate for the key repository, and one gate for the data repository. The generic application server holds a - 1 gates for the application repositories of

the other application servers; if $a = 16$, we have a total memory requirement for gate storage of 300 bytes.

We may conclude that the total memory requirements for key and gate storage in both a hierarchical network configuration and even in a configuration featuring a form of full pairwise application connectivity are low, and they are a negligible fraction of the overall memory resources of the network nodes. This is especially true for the generic application members, whose interactions with the application server and the other members of the same application can be fully supported by a total of three keys and two gates, with a total memory requirement of 100 bytes.

4.4. Gate manipulation

Gates are stored in memory together with ordinary information, in undifferentiated form. It follows that a node may well forge a gate referencing a segment in a different node from scratch, and then try to use this gate to access the contents of that segment. In fact, any illegitimate access attempt of this type is destined to fail.

Let M and N be two nodes, and k be a key shared by these nodes (k will be an application key, if M and N are both members of the same application, or a nonlocal key, if N is a server and M one of its clients, for instance). Let P_N be the set of the passwords associated with node N, and let us suppose that node M forges gate $G = (N, T)$ referencing a segment in the primary memory of N. To this aim, M will have to use an arbitrary value for validation field T, as it does not possess the local key lk_N of node N. Let us now suppose that M tries to take advantage of G, for instance, by executing protection primitive *readSegment*(k, G, *addr*), where *addr* is the address of an area in the primary memory of M. In the execution of this primitive, node N uses its own local key lk_N to decrypt the protection field T of gate G into pair (C, p) (see Section 2.3.2). Then, quantity p is compared with passwords p_R and p_{RW} in the set of passwords P_N to validate G. If passwords are large and sparse, the probability of a casual match is vanishingly low, and validation is destined to fail.

We wish to remark that password validation guarantees that any illegitimate attempt to use a gate in the wrong node is destined to fail. For instance, suppose that node M holds gate $G = (N, T)$ referencing a segment in the primary memory of node N. Suppose also that M forges a new gate $G' = (N', T)$ by associating the name of a different node N' with validation field T. When M tries to take advantage of G', by executing protection primitive *readSegment*(), for instance, the recipient node N' will decrypt T and validate the password contained in T by comparing it with its own password. Of course, validation is destined to fail.

4.5. Gate revocation

Gates can be freely moved and copied in memory, and a node that receives a gate is free to

distribute this gate to other nodes. This means that gates, once granted, tend to propagate throughout the network. A relevant problem is that of gate revocation. A node *N* that creates a gate for a segment *S* and grants this gate to a recipient node *M* should be in the position to retract the gate from *M* as well as from all the nodes that received a copy of the gate from *M*, recursively.

In our system, gate revocation can be obtained by taking advantage of passwords. Let P_N be the set of passwords associated with node *N*, and suppose that *N* changes the values of these passwords; so doing, all the gates that reference the segments in *N* are revoked (it will be no longer possible to use these gates for successful segment access). For instance, suppose that node *M* possesses gate $G = (N, T)$ referencing segment $S = (N, C)$ stored in node *N*. In order to take advantage of *G* and read the contents of segment *S*, *M* will issue protection primitive *readSegment*(*k*, *G*, *addr*), where *k* is a key shared by *M* and *N*, and *addr* is the address of an area in the primary memory of *M*. In the execution of this primitive, node *N* uses its own local key lk_N to decrypt the protection field *T* of gate *G* into pair (*C*, *p*). Then, quantity *p* is compared with the passwords in P_N to validate *G*. Of course, if the passwords were changed, this validation attempt is destined to fail. Despite its simplicity, this gate revocation mechanism possesses a number of interesting properties. It is [14]:

- *Transitive*, that is, if a node grants a gate for a segment in its own primary memory to a recipient node that in turn transmits gate copies to other nodes, the effects of the revocation propagate to all these copies, recursively, at any transition depth.
- *Temporal*, that is, the effects of the revocation can be reversed by taking advantage of the same mechanism used for the revocation (i.e. by restoring the original passwords).
- *Immediate*, that is, the holder of a given gate cannot use this gate immediately after the revocation. Indeed, execution of a *readSegment*() or *writeSegment*() primitive specifying a given gate as an argument is destined to fail as soon as the passwords are changed.

If we change the passwords of a given node, we revoke the gates for all the segments in that node. This gate revocation mechanism cannot be used to revoke the gates for a single segment selectively. A result of this type can be obtained by executing the *deleteSegment*() primitive and deleting the segment. As seen in Section 2.3, this primitive does not alter the contents of the memory area corresponding to the deleted segment. This means that it will be possible to allocate a new segment in the same memory area and proceed to a new distribution of gates for this new segment; distribution will involve only those nodes that should not be affected by the revocation. This form of revocation results to be *transitive* (the effects of the revocation propagate to all the copies of the gates referencing the deleted segment) and *immediate* (the revocation comes into effect as soon as the segment is deleted). It is even possible to take advantage of the fact that two or more segments can overlap in memory. In a situation of this type, we can have different gates for the same memory area. If we

delete one of these segments, we revoke the corresponding gate (and all the copies of this gate); revocation does not affect the validity of the gates for the other segments.

4.6. Considerations concerning security

4.6.1. The threat model

In a sensor network, a sensor node is an *internal adversary* if it is a network member, it is authenticated to the other sensor nodes, and is compromised. When a node is compromised, it is supposed to reveal its local key and any key shared with the other network nodes. The sensor node is an *external adversary* if it is not a network member, and uses different means of attack to reach the network [2].

An *external attack* carried out by an external adversary is called a *passive* attack if the external adversary eavesdrops packets, whereas the attack is *active* if the adversary injects or modifies packets. In an *internal attack*, an internal adversary behaves as a legitimate network member but in fact deliberately deviates from the specification of the intended application. In contrast with wired networks, in a wireless sensor network a simple radio transceiver is sufficient for an adversary to access the wireless medium and attempt an external attack. Internal attacks are facilitated by the fact that wireless sensor networks are often deployed over large, unattended, and possibly hostile areas, and sensor nodes typically lack adequate support to tamper-resistance.

With reference to this threat model and external attacks, the design of our distributed storage protection system has been aimed at fulfilling the following security requirements:

- *Secrecy*. It is infeasible for an external adversary to derive any piece of information from the messages eavesdropped in the network.
- *Authenticity*. It is infeasible for an external adversary to forge a call to a protection primitive or its results.
- *No-replay*. It is infeasible for an external adversary to replay a call to a protection primitive or its results.

As far as internal attacks are concerned, our protection system is intended to comply with the *forward security* requirement, i.e. a compromised node that generates an attack is logically evicted from the application as soon as the attack is detected. In the following, we shall assume the presence of an intrusion detection system [2] that monitors the network activities to detect possible compromised nodes.

4.6.2. Security analysis

We shall now analyze the security properties of our protection system in detail. More specifically,

we shall demonstrate that protection primitives *readSegment*() and *writeSegment*() satisfy the above-mentioned security requirements of secrecy, authenticity, no-replay, and forward security. As stated in Section 2.2, ciphers throughout this paper are supposed to comply with the Counter with CBC-MAC (CCM) encryption mode, which supports both authentication and confidentiality [11]. In the following, we shall refer to CCM for use with AES [37]. In fact, AES is generally considered a secure cipher [23], and several off-the-shelf sensor nodes, e.g. Tmote Sky [28], provide AES-128 encryption at the hardware level, with negligible overhead in terms of delay, storage, and energy consumption [8]. We shall assume that the key size is such as to discourage any form of exhaustive search, e.g. 128 bits, and that CCM encryption is secure if the underlying block cipher is secure. Informally, CCM security means that, for an adversary that has no access to the secret key, it is infeasible both to forge a valid ciphertext (*authenticity property*), and to derive any piece of information from a valid ciphertext (*secrecy property*) [18].

Figure 3 shows the communication protocols of *readSegment*() and *writeSegment*(). The protocols are illustrated with reference to the execution steps indicated in Section 2.3.2. E_M and E_N are the nonces generated by nodes *M* and *N*, *G* is a segment gate, *a* indicates the contents of the segment involved in the execution of the primitive, *k* is the cryptographic key used for communication between *M* and *N*, and finally, *reply* specifies whether execution of the primitive was successful, or not. For each given message, the figure specifies the execution step at which this message is transmitted. In the protocol of *readSegment*(), quantity *a* is enclosed in square brackets to denote that this quantity is empty if execution fails.

We are now in a position to demonstrate that *readSegment*() and *writeSegment*() are secure with respect to an external adversary. In both primitives, encryption of messages M3 and M4 in the CCM mode guarantees both message secrecy and authenticity with respect to an external adversary that does not know *k*. Several implications follow:

- *Secrecy*. An external adversary cannot derive any piece of information from the ciphertext contained in M3 and M4. In particular, the adversary cannot extract the segment gate *G* or the segment contents *a*.
- *Authenticity*. An external adversary cannot forge messages M3 and M4. In particular, the adversary cannot modify the segment contents *a* that are returned by *readSegment*() in message M4, and written by *writeSegment*() in message M3. Furthermore, the adversary cannot modify the final reply, returned by both primitives in message M4.
- *No-replay*. An external adversary cannot replay any operation call or result. This follows from the fact that, as seen in Section 2.3.2, the encryption is intended to indissolubly link the nonces with the gate and the segment contents.

M1	$M \rightarrow N$:	*nonce request*	(step 1)
M2	$N \rightarrow M$:	E_N	(step 2)
M3	$M \rightarrow N$:	$\{readSegment, G, E_N, E_M\}_k$	(step 3)
M4	$N \rightarrow M$:	$\{reply, E_M, [a]\}_k$	(steps 5, 6)

(a)

M1	$M \rightarrow N$:	*nonce request*	(step 1)
M2	$N \rightarrow M$:	E_N	(step 2)
M3	$M \rightarrow N$:	$\{writeSegment, G, E_N, E_M, a\}_k$	(step 3)
M4	$N \rightarrow M$:	$\{reply, E_M\}_k$	(steps 5, 6)

(b)

Figure 3. Communication protocols of protection primitives (a) *readSegment*() and (b) *writeSegment*().

Let us now consider an internal attacker participating in application *A* and running in node *N*. The attacker can take advantage of all the gates and the keys stored in *N* to issue illegitimate calls to the primitive operations. However, as soon as the intrusion detection system suspects that node *N* is compromised, the protection system starts up a rekey action to replace the application key and evict *N* from the application. In detail, the application server selects a new application key and distributes this key to all the nodes which are members of application *A*, except node *N*. To this aim, as seen in Section 3.2, the application server writes the new key into the key repositories of all the members of *A* except *N*. Then, the application server sends a rekey message; consequently, each node executes primitive *readSegment*() to read the new key from its own key repository, and then updates the key. For the given node, execution of *readSegment*() uses the nonlocal key shared between the application server and this node. As the compromised node *N* does not know this key, it cannot eavesdrop the new key.

4.7. Relation to previous work

A previous paper [25] demonstrates the possibility to take advantage of cryptographically protected pointers for segment access, with reference to a single-processor architecture featuring *ad-hoc* hardware for protection support. In that architecture, a set of *protections registers* are reserved inside the processor to contain protected pointers in plaintext. The instruction set is designed to comply with a memory address format that includes the specification of a protection register and an offset; the offset specifies a memory location in the segment referenced by the protected pointer contained in the protection register. The protection primitives are designed to be implemented at the hardware level as machine instructions; for a few protection primitives, software support is necessary, e.g. for memory management. Protection registers are mainly aimed at solving an important performance problem, i.e. the high cost, in terms of processor time, of a transformation of a ciphertext pointer to plaintext at each memory access.

In a subsequent paper [26], the protected pointer paradigm was extended with reference to multithreaded programs in a single address space environment supporting the notion of segmentation with paging. Protection is exercised at the level of a single page. A protected pointer references a segment and specifies a set of access rights for the pages that form this segment. The address translation circuitry performs the necessary validation of access privileges. Implementation is aimed at encapsulating the protection system, so that user processes are prevented from altering its intended behavior even if the design criteria and underlying algorithms are publicly known. The solution proposed exploits the usual separation between the two processor modes, a kernel, privileged mode and a user mode with restricted memory access.

In this paper, we have taken advantage of cryptographic techniques for the protection of memory pointers in a fully distributed, sensor network environment. A careful, thorough redesign of the protection system has been necessary to cope with the stringent limitations existing at the hardware level, and principally, a limited memory space with no provision for a protected memory area of the operating system kernel, and the lack of hardware support for the two processor modes, kernel and user.

5. CONCLUDING REMARKS

With reference to a distributed architecture consisting of sensor nodes connected in a wireless network, we have presented a paradigm of a protection system based on applications and segments. An application is the result of the joint activities of a set of nodes that cooperate in the same purpose. Segments are the basic unit of information gathering and transmission between the nodes. In our paradigm:

- A software routine running in a given node has unlimited access to the whole primary memory of that node, whereas access to a remote memory area in a different node can only be accomplished on a segment basis, by presenting a gate for the segment involved in the access. A gate is protected pointer that references a segment and specifies a set of access rights for this segment. The cryptographic form of gates in memory guarantees that any attempt to forge a gate from scratch or alter an existing gate (e.g. to amplify the access rights it contains) is destined to fail. A node holding a given gate can transmit this gate to another node, thereby granting the access rights specified by this gate to the recipient node.
- Two special node functionalities are supported, application servers and segment servers. An application server is used in each application for key management, rekeying, and information gathering. Segment servers are used for inter-application communication.
- The protection system defines three types of cryptographic keys: local keys, nonlocal keys and

application keys. Each node holds a local key to encrypt the gates for the segments in its own primary memory, a nonlocal key to communicate with the application server, and an application key to communicate with the other nodes of the same application. If a node is connected with a segment server, a further nonlocal key is necessary to communicate with this server. Nonlocal keys are also used for point-to-point communication between members of different applications.

- A small set of protection primitives forms the interface between the protection system and applications. These primitives allow a node to allocate segments locally, to generate gates for the local segments, and to use gates in remote segment accesses.
- The rekey mechanism is based on a segment, the key repository, which the application server reserves in its own primary memory for each application member. Key replacement is initiated by the application server that inserts the new key into each key repository and then sends a rekey message to all the members. Consequently, each member uses a gate for its own key repository to read the new key. The rekey mechanism takes advantage of key naming to cope with losses of rekey messages. This feature is especially important for reliable application rekeying in an unreliable network environment.
- We have considered two different network topologies in special depth, a configuration featuring full pairwise connectivity at the application level, supported by a point-to-point link between each pair of application servers, and a hierarchical topology featuring a general server that gathers data from all the application servers. In both cases, the total memory requirements for key and gate storage result to be a negligible fraction of the overall memory resources of the network nodes.

The idea of using different types of cryptographic keys, appropriate for different types of communication, is certainly not new. For instance, in [39], each node holds a global key shared with all the network nodes, a cluster key shared with all the neighboring nodes, an individual key shared with the base station, and a pairwise key shared with each of its immediate neighboring nodes. It has been reputed that key differentiation may facilitate key management and enhance overall system security [7]. In our system, different keys are used for point-to-point communication between different nodes, in the interactions between application members, and to compensate for the lack of hardware support at the node level for the two processor modes, kernel and user.

Fine-grained data access control is considered hard to obtain in an environment based on symmetric-key cryptography owing to the intrinsic complexity of key management [36]. We have obtained a result of this type by a careful protection system design, even within the stringent limitations existing in sensor nodes on hardware complexity and available hardware resources.

ACKNOWLEDGMENTS

This work has been partially supported by the EU FP7 Integrated Project PLANET (Grant Agreement no. FP7-257649), and by the TENACE PRIN Project (Grant no. 20103P34XC_008) funded by the Italian Ministry of Education, University and Research.

REFERENCES

[1] M. Anderson, R. D. Pose, C. S. Wallace, "A password-capability system," *The Computer Journal*, vol. 29, no. 1 (February 1986), pp. 1–8.

[2] I. Butun, S. D. Morgera, R. Sankar, "A survey of intrusion detection systems in wireless sensor networks," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1 (2014), pp. 266–282.

[3] H. Cha *et al.*, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 148–157.

[4] H. Chan, A. Perrig, D. Song, "Random key predistribution schemes for sensor networks," *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, Oakland, California, USA, May 2003, pp. 197–213.

[5] J. S. Chase, H. M. Levy, E. D. Lazowska, M. Raker-Harvey, "Lightweight shared objects in a 64-bit operating system," *Proceeding of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, Vancoover, October 1992; in: *SIGPLAN Notices*, vol. 27, no. 10 (1992), pp. 397–413.

[6] O. Cheikhrouhou, A. Koubâa, G. Dini, M. Abid, "RiSeG: a ring based secure group communication protocol for resource-constrained wireless sensor networks," *Personal and Ubiquitous Computing*, vol. 15, no. 8 (December 2011), pp. 783–797.

[7] X. Chen *et al.*, "Sensor network security: a survey," *IEEE Communications Surveys & Tutorials*, vol. 11, no. 2 (second quarter 2009), pp. 52–73.

[8] R. Daidone, G. Dini, M. Tiloca, "On experimentally evaluating the impact of security on IEEE 802.15.4 networks," *Proceedings of the 2011 International Conference on Distributed Computing in Sensor Systems and Workshops*, Barcelona, Spain, June 2011.

[9] G. Dini, I. M. Savino, "LARK: a lightweight authenticated rekeying scheme for clustered wireless sensor networks," *ACM Transactions on Embedded Computing Systems*, vol. 10, no. 4 (November 2011).

[10] A. Dunkels, B. Grönvall, T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," *Proceedings of the First IEEE Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004, pp. 455–462.

[11] M. J. Dworkin, *Recommendation for Block Cipher Modes of Operation: the CCM Mode for Authentication and Confidentiality*, Technical Report, National Institute of Standards and Technology, Special Publication 800-38C, May 2004, Gaithersburg, MD, USA.

[12] M. O. Farooq, T. Kunz, "Operating systems for wireless sensor networks: a survey," *Sensors*, vol. 11, no. 6 (2011), pp. 5900–5930.

[13] N. Ferguson, B. Schneier, *Practical Cryptography*. Indianapolis, Indiana: Wiley, 2003.

[14] V. D. Gligor, "Review and revocation of access privileges distributed through capabilities," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 6 (November 1979), pp. 575–586.

[15] V. C. Gungor, G. P. Hancke, "Industrial wireless sensor networks: challenges, design principles, and technical approaches," *IEEE Transactions on Industrial Electronics*, vol. 56, no. 10 (October 2009), pp. 4258–4265.

[16] G. Heiser, K. Elphinstone, J. Vochteloo, S. Russell, J. Liedtke, "The Mungi single-address-space operating system," *Software – Practice and Experience*, vol. 28, no. 9 (July 1998), pp. 901–928.

[17] F. Hu, N. K. Sharma, "Security considerations in ad hoc sensor networks," *Ad Hoc Networks*, vol. 3, no. 1 (January 2005), pp. 69–89.

[18] J. Jonsson, "On the security of CTR + CBC-MAC," *Proceedings of the 9th Annual International Workshop on Selected Areas in Cryptography*, St. John's, Newfoundland, Canada, August 2002, pp. 76–93; in: *Lecture Notes in Computer Science*, vol. 2595.

[19] F. Kausar, S. Hussain, J. H. Park, A. Masood, "Secure group communication with self-healing and rekeying in wireless sensor networks," *Proceedings of the 3rd International Conference on Mobile Ad-Hoc and Sensor Networks*, Beijing, China, December 2007, pp. 737–748.

[20] R. Kumar, E. Kohler, M. Srivastava, "Harbor: software-based memory protection for sensor nodes," *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, Cambridge, Massachusetts, USA, April 2007, pp. 340–349.

[21] R. Kumar, A. Singhania, A. Castner, E. Kohler, M. Srivastava, "A system for coarse grained memory protection in tiny embedded processors," *Proceedings of the 44th Annual Conference on Design Automation*, San Diego, California, USA, June 2007, pp. 218–223.

[22] H. Kurnio, R. Safavi-Naini, H. Wang, "A secure re-keying scheme with key recovery property," *Information Security and Privacy*; in: *Lecture Notes in Computer Science*, vol. 2384/2002, pp. 1–51.

[23] Y. W. Law, J. Doumen, P. Hartel, "Survey and benchmark of block ciphers for wireless sensor networks," *ACM Transactions on Sensor Networks*, vol. 2, no. 1 (February 2006), pp. 65–93.

[24] P. Levis *et al.*, "TinyOS: an operating system for wireless sensor networks," in: *Ambient Intelligence*. New York: Springer-Verlag, 2005, pp. 115–148.

[25] L. Lopriore, "Encrypted pointers in protection system design," *The Computer Journal*, vol. 55, no. 4 (April 2012), pp. 497–507.

[26] L. Lopriore, "Protection structures in multithreaded systems," *The Computer Journal*, vol. 56, no. 4 (April 2013), pp. 478–496.

[27] L. Lopriore, "Password capabilities revisited," *The Computer Journal*, first published online November 11, 2013; doi: 10.1093/comjnl/bxt131

[28] Moteiv Corporation, *Tmote Sky Low Power Wireless Sensor Module*. Datasheet, San Francisco, CA, USA, November 2006.

[29] T. Pham, P. A. Watters, "The efficiency of periodic rekeying in dynamic group key management," *Proceedings of the Fourth European Conference on Universal Multiservice Networks*, Toulouse, France, February 2007, pp. 425–432.

[30] V. Potdar, A. Sharif, E. Chang, "Wireless sensor networks: a survey," *Proceedings of the International Conference on Advanced Information Networking and Applications Workshops*, Bradford, United Kingdom, May 2009, pp. 636–641.

[31] S. Rafaeli, D. Hutchison, "A survey of key management for secure group communication," *ACM Computing Surveys*, vol. 35, no. 3 (September 2003), pp. 309–329.

[32] M. A. Simplício, P. S. L. M. Barreto, C. B. Margi, T. C. M. B. Carvalho, "A survey on key management mechanisms for distributed wireless sensor networks," *Computer Networks*, vol. 54, no. 15 (October 2010), pp. 2591–2612.

[33] M. Stamp, *Information Security: Principles and Practice*, 2nd Edition. Hoboken, New Jersey: JohnWiley & Sons, 2011.

[34] Y. Xiao *et al.*, "A survey of key management schemes in wireless sensor networks," *Computer Communications*, vol. 30, no. 11–12 (September 2007), pp. 2314–2341.

[35] J. Yick, B. Mukherjee, D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12 (August 2008), pp. 2292–2330.

[36] S. Yu, K. Ren, W. Lou, "FDAC: toward fine-grained distributed data access control in wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 4 (April 2011), pp. 673–686.

[37] D. Whiting, R. Housley, N. Ferguson, "Counter with CBC-MAC (CCM)," Request for Comments 3610, Internet Engineering Task Force, September 2003. Available (January 2015) at: *http://www.rfc-base.org/txt/rfc-3610.txt*

[38] J. Zhang, V. Varadharajan, "Wireless sensor network key management survey and taxonomy," *Journal of Network and Computer Applications*, vol. 33, no. 2 (March 2010), pp. 63–75.

[39] S. Zhu, S. Setia, S. Jajodia, "LEAP+: Efficient security mechanisms for large-scale distributed sensor networks," *ACM Transactions on Sensor Networks*, vol. 2, no. 4 (November 2006), pp. 500–528.